**Evaluating Lehman’s Laws of software evolution using the GitHub API**

**A dissertation submitted in partial fulfilment of**

**the requirements for the degree of**

**MASTER OF ENGINEERING in Computer Science**

**In**

**The Queen's University of Belfast**

**by**

**Jordan McDonald**

**\*ADD SUBMISSION DATE HERE\***

**\*** **Declaration Cover Sheet ADD\***

**\*Add acknowledgements here\***

1. **ABSTRACT**

This paper studies the validity of Lehman’s laws of software evolution when applied to one hundred open source projects hosted on GitHub. The data set that will be used to investigate this objective will be extracted from the GitHub API and focuses on the repository level which provides the novelty to this study. Metrics attained from the API have been extracted and attached to each law in turn as a means of quantifying the analysis and enabling the various hypothesis to provide insight into the validity of each law in this context.

…Add some more info on the conclusions\*\*

1. **INTRODUCTION**

The term software evolution represents the change of a software system as time progresses, factors that instigate this change include various forms of maintenance which can be categorised as adaptive, perfective, corrective and preventative[15]. To evaluate this change Lehman and Belady formulated the laws of software evolution, which attempted to outline the factors that drive growth and development of software, while also taking into account forces that lead reduced progress. Lehman theorised that most software is subject to change over the course of its existence and this change can be driven by a multitude of different events. The goal was to identify a set of laws that these changes would obey, or must obey in order for software to survive (Section 2.1).

The goal of this paper is to examine these laws in the context of open source projects hosted on GitHub, with a dataset mined from the GitHub API as the focal point for the study. GitHub is a hosting website designed for collaboration on a centralised repository of source code. Any user of the website can ‘Clone’ any public repository and read or alter the code, this serves as the backbone of modern open source development and helps facilitate the ‘fork and pull’ model of development. In addition to code hosting, collaborative code review, and integrated issue tracking, GitHub has integrated social features. Users are able to subscribe to information by “watching” projects and “following” users, resulting in a feed of information on those projects and users of interest. Users also have profiles that can be populated with identifying information and contain their recent activity within the site [2].

As of 2015, GitHub reports having over 9 million users and over 21.1 million repositories [3] making it the largest host of source code in the world [4]. This represents a period of rapid growth considering in 2010, announced on the official GitHub blog it was revealed that one million repositories were hosted on GitHub. These factors in tandem with the accessible GitHub API’s data on commits, code churn, issues, watchers and pulls among other metrics provide an excellent foundation to examine Lehman’s laws in a lesser analysed context and at a volume which in my current knowledge has not been addressed fully on another study.

This paper will perform a large scale analysis of open source projects hosted on GitHub, extracting data at the repository level in order to determine if Lehman’s laws hold or are contradicted by the findings. Each law will be represented by metrics taken from the API and the evolution of these metrics over time will provide an insight into software growth patterns, which in turn shall test the validity of the laws devised by Lehman.

1. **BACKGROUND AND RELATED WORK**
   1. **Background**

Initially devised in 1974 Lehman’s laws have undergone multiple changes as the years have progressed, with the latest alteration taking place in 1996. In his 1980 article [5] Lehman qualified the application of such laws by distinguishing between three categories of software:

* An S-program is written according to an exact specification of what that program can do.
* A P-program is written to implement certain procedures that completely determine what the program can do (the example mentioned is a program to play chess).
* An E-program is written to perform some real-world activity; how it should behave is strongly linked to the environment in which it runs, and such a program needs to adapt to varying requirements and circumstances in that environment.

It is evident that the laws reflect the E-program definition devised by Lehman, the emphasis on feedback and adaptations of software are key components of evolution. Each project in this study will in turn reside under the E-program umbrella and each law is applicable to this category, see below for a summary of each.

* **(1974) "Continuing Change"** - an E-type system must be continually adapted or it becomes progressively less satisfactory
* **(1974) "Increasing Complexity"** - as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it
* **(1974) "Self-Regulation"** - E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal
* **(1978) "Conservation of Organisational Stability (invariant work rate**)" - the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime
* **(1978) "Conservation of Familiarity"** - as an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
* **(1991) "Continuing Growth"** - the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime
* **(1996) "Declining Quality"** - the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes
* **(1996) "Feedback System"** (first stated 1974, formalised as law 1996) - E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base
  1. **Related Work**

Attempts at general data mining from GitHub has been prominent in recent years, Kalliamvakou et al [2] published a paper that highlighted the ‘promises and perils of mining GitHub’. This paper has a focus on avoiding common pitfalls in GitHub mining and concluded that there is valuable data to be found if these are avoided. M.M. Mahbubul Syeed [11] has previously performed a systematic literature review into the evolution of open source projects, the authors examine the data sets utilised, sources of the data and research trends in recent years. The author found that Lehman’s laws do not hold in certain cases, with individual laws in the research yielding contradicting results in regards to open source projects.

Additional papers have provided much more focused studies, Jyoti Sheoran et al [7] investigate the watcher mechanic on GitHub, which provides notifications to user who watch a repository each time an event occurs such as a commit or creation of an issue. The paper hones in on the contributors of a project, tracking to process of a user becoming a watcher to finally contributing to a project, finding that this process accounts for a huge bulk of the tested projects eventual contributors. Another study on this topic was conducted by Xu Ben et al [9] which performed visualisation on metric related to commits, low level code statistics and lines of code on a single project, this restriction limits the usefulness of the research. Georgios Gousios et al [4] look in depth at the GitHub ‘fork and pull’ model of development on a sample of 291 projects. The metrics utilised are among the widest ranging in previous literature, considering feature sets for the pull request itself, the project and the developers involved. An analysis was made on what projects utilise this model, the turnover rate of pull request and why requests are rejected. [11] Provides insight into what constitutes a projects popularity on GitHub using the starring mechanic, the paper theorised that this could be tracked over time to show the evolution of popularity. [13] Analyses issues (bugs) as part of open source software, correlating the data with watchers, forks and other metrics.

A similar study to that presented in this paper in regards to evolution was performed by Jesus M. Gonzalez-Barahona et al [8] was conducted on a long running FLOSS project, glibc inside a SCM repository with over 20 years of history. The paper also approaches the research through reference to Lehman’s laws. The metric utilised has a focus on commits, lines of code and files changed to represent evolution – a downside to this study is single project focus, this paper hopes to consider a much larger dataset in order to draw novel findings. [17] [18] take a single and seven project approach respectively with a focus on long running projects such as SQLite and the open source browser Firefox. [19] Has a sample size of nine projects and utilises code level metrics such as KLOC [10] also delves into software evolution and Lehman’s law, however from the context of databases.

**3.3 Novel approaches in this paper**

On conclusion of the literature review gaps in the research were identified from which novel contributions to the field could be made. Evaluating Lehman’s laws according to data from the GitHub API has not yet been fully investigated. This paper plans to represent each law with relevant metric and quantify the evolution of these data points. Prior studies that are similar to the approach in this paper have flaws – A) only investigating one project B) looking at evolution from the stand point of databases. This study will encompass a large data set with variation in the language of choice for the repositories, from this it will be possible to determine if a pool of different programing languages will support or contradict Lehman’s laws.

1. **PROPOSED METHODOLOGY**

**4.1 Research Questions & Hypotheses**

In order to provide scope to the research presented in this paper it is critical to set clear and defined research questions. Research question one will focus on the validity of Lehman’s laws in the context of open source GitHub projects, with multiple hypotheses that will attempt to draw out the relationship between each law and the metrics extracted from the API. A caveat of law three that has to be considered is a reduced scope due to the huge amount of possible metrics available that need considered, in this case we have restricted it to three data points.

**RQ1** - Is it possible using data extracted from the GitHub API to determine if OS software evolution over time reflects Lehman’s laws?

**H1** – If the amount of commits decreases the amount of star gazers will also reduce (law 1 + 6)

**H2** – Total lines of code increases as software system evolves (law 2)

**H3** – Issues, additions and deletions over time for will be normally distributed (law 3)

**H4** – As software evolves changes to lines of code should remain invariant over time (law 4)

**H5** – As Lines of code increases the amount of issues will also increase (law 5)

**H6** - Project issues will increase as code churn decreases (law 7)

**H7** – As the number of issue comments increases the number of issues should decrease (law 8)

**4.2 Project Selection**

To provide scope to the research performed in this paper, a process of identifying the volume and variation of the projects attained from GitHub needs to be defined - figure 1 demonstrates the selection process. The ten programming languages of choice have been chosen based on a ranking system seen in the GitHub blog post [14] which shows the top ten used languages (based on total active repositories) on the site in public and private repositories (excluding forks) as of August 2015.

Top 10 languages on GitHub [14]

1. JavaScript
2. Java
3. Ruby
4. PHP
5. Python
6. CSS
7. C++
8. C#
9. C
10. HTML

Select another programming language

Search each language for the most popular project (sorted by stars)

Identify the top ten languages

If total projects for the current language equals ten?

Has the current project been on GitHub for five years or more?

Does the project have a fifty percent plus affinity to the target language & metrics fully populated?

Add the project to the selection for analysis

Figure 1 - Flow chart showing the project selection progress

It is crucial to apply restrictions to the projects selected for each programming language in order to visualise the evolution of the software effectively and maintain the integrity of the target programming language requirement. The GitHub advanced search facility on the site allows the descending ordering of the ‘most stars’ for a programming language, each sequential project is then evaluated against two criteria.

1. Duration of project life on GitHub, with a set five year threshold which is chosen to ensure evolution can be mapped over a sustained period of time.
2. It is very common for most projects to use multiple programming languages, however GitHub allows users to examine a project for the breakdown of languages utilised. Using this each project prior to analysis has to meet the 50% target language affinity requirement.

This process will be applied to two hundred projects in total, the final dataset of one hundred will then be randomly selected with the intent of taking ten projects from each programming languages group of twenty.

**4.3 Data Collection**

GitHub provides a robust API which is ideal for mining the data associated with a project. The current version of the API is version three and all requests are performed over HTTPS, the data is returned in a JSON format which allows simplistic parsing of the metric required. Disadvantages to the API include the pagination system which restricts the amount of data that can returned in one request, which may lead to multiple similar requests taking place. The method utilised to collect this data will be AJAX as implemented in the JQuery JavaScript library, then once processed stored in MongoDB database.

**4.4 System Design**

To enable the research a workbench has been devised which will handle the automated collection of the data for each of the one hundred projects and to execute the statistical functions. The interact with the GitHub API the JQuery library will be leveraged in order the extract the relevant data via a HTTP call using Ajax, the response from the API will take the form of JSON which can then be parsed as required. In order to answer the hypotheses various statistical methods would need to be applied, to handle this the R environment was integrated in the workbench in order to reliably get results using the built in libraries of R.

**4.4.2 Overall System Architecture**

URL(s)

Webpage

GitHub API

Raw data

JSON data

JSON extractor module (JS)

MongoDB

Java Servlet

DB Query

R Environment

Figure 2 – shows the general system processes

**4.5 Data Analysis Methods**

Now it would be prudent to discuss the structure of the parsed data, each metric is associated with an accompanying time series that signifies the start of a weekly interval. The dataset itself is organised into a vector with each point containing weekly counts of the frequency of the metric in that particular time period. some data points may have gaps between frequencies that exceed the weekly structure, therefore padding has been introduced to fill the gaps in a project as required, in this case each padded weekly interval will be assigned a zero to signify no activity in that period. To ensure the integrity of the research the first six months for each projects have been ‘trimmed’ this is to account for projects that have origins that outlive the GitHub platform. The reason for this is to remove the possibility of initial ‘dump’ of data from a pre-existing polluting the results with the potential for significant statistical outliers. The metrics that will be extracted from the API in order to quantify the analysis are listed below, the relationship between these and the hypotheses has been covered in a previous section (with additional metrics added for the flexibility of the workbench).

* **Stargazers** - Repository Starring is a feature that lets users bookmark repositories. Stars are shown next to repositories to show an approximate level of interest [16].
* **Commits** - A commit, or "revision", is an individual change to a file (or set of files).
* **Additions & Deletions** – represent modified, added or removed lines of code.
* **Issues** - Issues are suggested improvements, tasks or questions related to the repository.
* **Fork** - A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes.
* **Commit Comments** – Messages that a user has attached to a specific commit.
* **Tags** – Often created when a new version of the project is released.
* **LOC** – total lines of code at a certain point
* **Growth Rate** – how much a certain metric changed per time interval

**4.5.1** – Statistical Methods

**4.5.1.1** **Growth Rate**

This equation has significant value in the context of software evolution, where values are analysed over a period of time. In particular in tandem with an LOC metric to answer to hypothesis two which is an ideal use case for this statistic. In addition to this it will have value when applied to hypothesis five, but in this case it will be mapped to a time series and cross correlation with issues. This statistic will be utilised to generate a percentage showing the amount of growth.

1. Growth Rate -
2. Average Growth Rate –

X = current value

Y = past value

N = total samples

**4.5.1.2 Shapiro Wilks Test**

This particular test will be applied to the three metrics stated in hypothesis three in order to determine the distribution of the data and evaluate the normality. This particular statistic utilises the null hypothesis principle (The null-hypothesis of this test is that the population is normally distributed) using a set alpha (0.05 in this case) if the p value is below this threshold then the null hypothesis is rejected and there is evidence that the data tested is not from a normally distributed population

**4.5.1.3 Cross Correlation**

To adequately answer hypotheses one, five, six and seven a cross correlation will be performed which will quantify the relationship between two time series by identifying lags of series x that will be useful predictors of series y. In the case of this research, multiple lag values will be considered to determine if a change in one metric weeks prior will have an impact on a series weeks in the future, in other words to determine if x leads y.

**4.5.1.4 Variance & Standard Deviation**

Law four concerns itself with an invariant work rate, this can be interpreted to applying a variance on the growth rate of the projects LOC. The growth rate will become a series of growth rate values between each weekly LOC, the variance will be applied to this series and a medium operation will be applied to the priori generated one hundred variances in order to facilitate discussion.

**5. RESULTS**

**5.1 – Hypothesis One**

The results generated for each hypothesis will now be examined in sequence, HP1 which represents laws one and six will be initially examined. A lagged cross correlation was performed with multiple different values in order to determine if and when the impact of making a change i.e. a commit will have a direct effect on stargazers and in particular what duration is of time after a commit is the change felt most significantly. The results of this experiment are shown in figure three which shows the results with a lag ranging from -9 to no lag applied.

|  |  |
| --- | --- |
| Lag Amount | Percentage of positive correlations |
| 0 | 60% |
| -1 | 61% |
| -2 | 57% |
| -3 | 60% |
| -4 | 60% |
| -5 | 54% |
| -6 | 55% |
| -7 | 55% |
| -8 | 50% |
| -9 | 51% |

Figure 3 – percentage positive cross correlation at different lags

The results presented in figure three show a clear relationship between the amount of the lag applied to the commits and the percentage of positive correlations that have been attained between the lagged commit count and the present stargazer count. As the lag is increased (in this context each increment represents the count of commits a week further into the past) the amount of correlation begins the decrease which indicates that the further apart the commit frequency in a particular week from the present stargazer count the less impact it will have on the amount of stargazers. It is possible that in the case of extreme lag applied that the effect of that change has already been felt at some point in the interim, therefore it may have already changed the count of the stargazers in a positive or negative way. If we now consider the inverse of this trend it appears that if changes in the amount of commits contributed to the project are recent (0 lag to -4 lag) the amount of stargazers is more likely to correlate which would suggest that the amount of commits made recently has a greater bearing on the number of stargazers than those which typically happened over a month prior. If we consider this from a potential stargazer’s point of view it stands to reason that they will be more likely to ‘star’ or ‘unstar’ the project based upon the recent changes that have been made to the system rather than those that happened in points in time beyond a few weeks due to having a greater investment in commits that have more immediate effects on the project.

The next step will be to consider the significance of the percentage value towards accepting or discarding the hypothesis. The value itself for all lags is not conclusive enough to be able to determine this, however an argument could be made that the lesser lag values support the hypothesis. In particular the -1 commit lag which is the best performing correlation percentage with stargazers which indicates that the optimum time is week before the stargazers react to the commit count and decide whether to remain stargazers or to stop following the project. To support a conclusion figure four has been provided which shows the distribution of each project correlation coefficient in each of the examined lag permutations. Based on these graphs it becomes conclusive that the hypothesis can be rejected due to the almost random distribution of the cross correlation values which show only a minor affinity towards positive correlations as the lag is reduced.

In conclusion using the evidence available, laws one and six of Lehmann’s laws of software evolution do not hold for the open source projects hosted on GitHub. Law one and six both state that in order to maintain user satisfaction the project will need to continually change and grow to maintain user satisfaction. A reason why this does not apply to the context of the GitHub platform could be attributed to the starring process which serves as a repository ‘bookmark’ for the user to show an level of interest that does not extend to receiving notifications etc. about the project. This would suggest that independent of the amount of commits (change) made the user will continue to remain starred until they have a reason to change that stance (become less satisfied)/stop supporting the project which highlights a clear disconnect between these particular laws and the GitHub platform.

Figure 4 – graphs showing the distribution of correlation values for each of the 100 projects on different lags intervals

**5.2 Hypothesis Two**

In order to derive conclusions to the hypothesis that represents the second law "Increasing Complexity" first complexity had to be defined according to the metric available using the GitHub API. LOC was chosen as an appropriate measure and as before the lines of code each week was organised into a vector and applied to a growth rate algorithm which determined the average percentage growth for each week from the first and last week’s LOC total. Each of the one hundred projects was subject to this process which resulted in the generation of one hundred growth rate values from which the validity of the hypothesis could be evaluated by determining the percentage of projects that increased in size over time.

Figure five visualises the results of this process, the majority of the projects do increase in size as the software system evolves. This is generally to be expected as time progresses the demand for new features and functionality to improve on the existing software will be constant in order to maintain a user base, this is particularly crucial in open source software where new libraries and technologies are introduced at a rapid frequency. However there remains several projects that have confounded the hypothesis and reduced in size, law two states that this could be the side effect of work being done to actively reduce or maintain the size of the project. Reasons that this could occur is refactoring, which is a prominent part of software evolution and certain projects may have taken steps to streamline or alter the architecture of the system. Upon investigation of the seven projects that decrease in size no particular pattern could be identified in terms of programming language or other factors so an assumption could be made that the reasons discussed prior could account for this.

In conclusion the evidence would suggest that law two holds as it provides scope for projects to actively minimise the complexity which could be attributed to refactoring and removing functionality among other factors. When hypothesis two is observed there is a significant enough of a majority showing positive growth to indicate that it also holds based on the evidence attained.

Figure 5 – shows the amount of projects whose LOC increased or decreased over time

**5.3 Hypothesis three**

To capture the essence of the third law three metrics would have to be considered to represent the ‘products and process measures’ and the ‘self-regulating’ keywords, in this case additions/deletions in tandem with issues was chosen. In order to determine if these measures were close to normal the Shapiro-Wilks test of normality was leveraged for each metric extracted from the one hundred projects. In order to determine the significance of the measure the p-value was utilised which could then be compared to an alpha (0.05) to determine if the null hypothesis (the population is normally distributed) could be rejected, from this percentages could be generated showing the amount that are within the threshold.

Figure six shows the overall results of this process, deletions and additions for each of the one hundred projects are all rejecting the null hypothesis ad therefore not from a normal distribution. This reflects the nature of open source development in which changes to the master branch can be made dynamically at any time, as a consequence of this it is possible that there will be periods where no change to the code is made. As a result of this the amount of additions and deletions may fluctuate from week to week with no consistency in the amount of code change, depending on the nature of the change which could vary from a minor bug fix to integrating a new feature. Once issues are observed the result is not as conclusive, this could be down to factors such as

a) The volume of issues per week typically is much lower than additions and deletions which would reduce the scope for the same extremes of fluctuation.

b) Issues in GitHub terminology could also be opportunities to refactor/improve the code base and as time progresses and more features are added to the software it is likely that issues would continually be identified by the users or development team.

However even when these points are considered the majority of the projects issues are not normally distributed which would indicate that the hypothesis and the law itself are refuted due to the overwhelming evidence provided. The reason for this once again is a product of the open source paradigm which thrives upon contribution from distributed collaborators at any point in time, pull requests are monitored by the core projects team but a change is reviewed and accepted at any arbitrary point in time which disrupts the normality of particularly additions and deletions, which is a key driver of challenging this law.

|  |  |  |
| --- | --- | --- |
| Percentage of Issue | Percentage of Deletions | Percentage of Additions |
| 93.548386 | 100% | 100% |

Figure 6 – the percentage of p values for each metric that are significant

**5.4 Hypothesis four**

In order to determine an invariant work rate LOC was chosen as the measure which was then applied to the growth rate algorithm which measured the amount of weekly growth at each point of the projects life span in order to generate a vector of percentage growth rate values. The variance of each vector was then extracted in order to determine how much of an instability in ‘work rate’ was present in each project, see figure seven to view the distribution of variance for each of the one hundred projects. From these graphs it is clear that the growth rate variance for each project can fluctuate between different extremes, the highest values are representative of projects whose growth is unpredictable, possibly due to sudden significant shifts in growth or may have long periods with no change to growth rate that precede an a spike in contributions. It is possible to observe significantly outliers that are prominent in the set of variance values, therefore to aid in interpretation the median of these values was calculated – **30.290**.

Based on the graphs it is difficult to determine an outcome to the hypothesis, to represent a reasonable invariant growth rate the standard deviation for each projects growth rate vector was calculated. Since the amount of lines of code that change per weekly interval may vary based on a number of factors, introducing the standard deviation as to act as a threshold to determine a reasonable distance from the mean would prove useful. This measure would enable determining the percentage of growth rate values are within one standard deviation distance from the mean growth rate value for each project – see figure eight to view the results of this process. The vast majority of each projects vectors are showing significant affinity to the one standard deviation invariant work rate threshold which suggests that the over the course of the projects life cycle the lines of code changes remain within a reasonable level of invariance. However this does not account for the growth rate values outside of the threshold which may represents growth that is among the more extreme cases, however it is reasonable to assume that over the course of a systems life span there will be changes that are more significant than the norm.

In conclusion based on the metrics and evaluation measures utilised there is no possible way to accept the premise of the hypothesis due to the large amount of variation shown in each project and when the standard deviation was considered a subset of a large portion of projects growth rate values were not invariant over time. In this particular scenario Lehmann’s fourth law of “Conservation of Organisational Stability (invariant work rate)" is disputed.

Figure 7 – distribution of LOC growth rate variance for each of the one hundred projects

Figure 8 – % of each projects growth rate values within one standard deviation

**5.5 Hypothesis Five**

Law five "Conservation of Familiarity" suggests that excessive growth off software as time progresses will reduce the mastery of the user base and lead to reduced satisfaction. To measure this the growth rate of lines of code has been chosen to represent growth and issues has been utilised as an indicator of user bases mastery of the software which can then be applied to a cross correlation to determine if a positive correlation is present. Figure nine shows the results of this process by calculating the percentage of cross correlation values for each project at different lag points that show a positive correlation.

|  |  |
| --- | --- |
| Lag Amount | Percentage of positive correlations |
| 0 | 50% |
| -1 | 50% |
| -2 | 49% |
| -3 | 46% |
| -4 | 51% |
| -5 | 44% |
| -6 | 41% |
| -7 | 44% |
| -8 | 50% |
| -9 | 45% |

Figure 9 - % of cross correlation values for each project showing a positive correlation

Initially a discussion will be made on the impact of applying a lag to the LOC growth rate has on its correlation with the amount of issues generated by users for the projects. Based on particular the negative eight lag result in comparisons to those which represents the impact of a change in LOC in weeks closer to the ‘present’ point for issues there appears to be no point that an increase/decrease in growth rate has an impact on the amount of issues. Reasons for this could include the sporadic nature of growth in open source projects which often do not confirm to a schedule, new code is often integrated on an ad-hoc basis and if a change introduces any problems (an issue) it may only become evident in a very specific use case at an arbitrary point in time before being reported. In addition to this Lehmann focuses on ‘familiarity’ and ‘mastery’ which are not terms which cannot be applied to the dynamic potential pool contributors outside of an OSS project core development team who may develop the code base without in depth knowledge about the intricacies of the software. Therefore may unknowingly introduce issues that may only be discovered at some point in the future or by having a fresh perspective on the code base could discover potential issues that the core team did not consider.

The next step will be to interpret the percentage results that have been obtained at applying the cross correlation to varying growth rate lags. Overall the results do not show any relationship between the amount growth rate and the amount of issues and mostly highlights a random distribution of cause and effect in this case. Only one of the outcomes produce a majority positive correlation for each of the one hundred projects, based on this evidence the hypothesis will be rejected and in turn law five is refuted based on this dataset and interpretation. To reinforce this point a series of graphs has been presented in figure ten which highlight the lack of relationship and random nature of the correlation values attained.

Figure 10 – distribution of cross correlations at different lags

**5.6 Hypothesis Six**

In interpreting this law to the GitHub API quality will be measured by the number of issues that occur in each weekly interval and LOC in the same structure will be used to represent code churn/the system being maintained and adapted. To determine if a decrease or stagnation in the lines of code will lead to an increased number of issues (or vice versa) in the set of projects a cross correlation was again applied with various lag parameters tested to supplement the analysis. The main target was to evaluate each generated correlation value and count the amount of times for each of the one hundred projects that a negative correlation occurs, this has been expressed as a series of percentages in figure eleven.

|  |  |
| --- | --- |
| Lag Amount | Percentage of negative correlations |
| 0 | 38% |
| -1 | 36% |
| -2 | 37% |
| -3 | 37% |
| -4 | 35% |
| -5 | 35% |
| -6 | 32% |
| -7 | 33% |
| -8 | 33% |
| -9 | 32% |

Figure 11 – percentage of correlations for different lags that are negative

To begin the discussion the impact of applying an increasing amount of lag to the lines of code will be reviewed, a clear pattern is evident which shows the overall percentage decreasing as the LOC lag is moved further into the past. This indicates that increasing or decreasing the LOC of a project will more likely have an immediate effect on the amount of issues than a change that happened many weeks prior, which in turn suggests the amount of issues is dependent on recent changes to the lines of code. Logically this makes sense as introducing new features in the past may typically spawn issues that were not immediately evident to the core team and make takes weeks to come to the surface following extensive usage and feedback from the user base. This would explain why the amount of positive correlations increase as the lag is increased and as an inverse negative correlations increase the closer in time to the present as the LOC has been altered to solve the problem or it has stagnated and the problem still remains and may have contributed to further issues.

If the percentages themselves are considered it indicates that the amount of negative correlations in this context is the minority result, rather than changes in lines of code decreasing the amount of issues in most cases the amount of issues increase (or rather than stagnation/decrease in LOC introducing more issues, it reduces the amount of issues). This brings up a facet of open source development that contributes this phenomenon, typically a subset of the core team reviews pull requests and decides on whether to merge an alteration to the code base. This potentially isolates a large proportion of the contributors who have a more ad-hoc presence who could foresee a bug/future issue that could come as a result of accepting a certain merge. Therefore it is likely that in hindsight after the pull request was accepted and it has been extensively utilised by the user base issues could arise after an arbitrary amount of time, the table in figure eleven suggests that changing the LOC further into the past have a greater impact towards issues than immediate changes. Overall this suggests that an increase to lines of code is more likely to spawn an issue than stagnation or a decrease, this could be a result of introducing new features which could have only been tested in isolation by the core team and when exposed to the public more issues arise. Whereas a reduction in LOC could be a result of refactoring or removing dead code, therefore improving the software and preventing future problems that may arise.

To conclude reference should be made back to the hypothesis to determine an outcome, the results in figure 11 provide evidence which directly refutes the law using this dataset despite it holding in some cases. Therefore it becomes prudent to examine not only a binary decision of positive/negative correlation to distribution could also be examined, see figure twelve which shows a distribution of values that appear random which indicates the behaviour as defined by Lehmann for this law is not being adhered to.

Figure 12 – distribution of correlations at different lag points

**5.7 Hypothesis Seven**

**6. THREATS TO VALIDITY**

In this section discussion will be made about the papers approach in order to determine areas from which the findings can be scrutinised – initially in the context of construct validity. Initial hypothesis generation will be examined, due to a focus on the metrics that can be attained from the GitHub API Lehmann’s laws had to be interpreted into hypotheses that represent the intent of each law as accurately as possible. In some cases logical metrics were available such as using stargazers to measure ‘satisfaction’, however in other cases there is room for dispute. An example of this is evidenced in law two ‘increasing complexity’ this study represents complexity as lines of code, however it is also possible to choose more appropriate measures such as McCabe’s cyclomatic complexity which would involve delving into lower level metrics at the code base, which is beyond the scope of this study. In addition to this law six focuses on quality, the metric that has been attached to this law is issues and its relationship with code churn (additions and deletions) but in reality this is a much more abstract term that could account for testing code coverage, architecture, count of bugs among others but due to the restrained of utilising only API produced data, this was a good option that captured the essence of the law which was the main goal when generating hypotheses.

The pre-processing of the dataset also has the potential to impact the validity of the results, the first six months of each data point is trimmed from the evaluation to account for projects migrating to GitHub and the initial dump of data associated with this process. This process of indiscriminate of the whether a migration has occurred or not, so projects who have spent their entire life span on GitHub will also be targeted, this directly removes the possibility of analysing the early stages of evolution for these particular projects.

It should also be noted that the rate of activity on each project has not been a deciding factor in the selection process. Therefore it is possible that among the range of projects there will be some that are maintained much more effectively than others, this is dependent on factors such as the size of the team actively working on the project and the amount of general user collaboration on GitHub. This might lead to cases where the activity of the team itself becomes a driver of software evolution which this study does not account for and could be an avenue for future work.

Threats to the external validity of the findings also will need to be examined, particularly if the results from this paper can be generalised to open source projects on GitHub in general. Despite the selection of a fairly large set of projects there is no evidence to suggest that the results will remain consistent when applied to a totally different dataset, however due to the paper targeting the most popular projects on GitHub it can be seen as representation of typical open source development for well supported projects not necessarily those that have reduced attention from users.

\*discuss conclusion validity

**7. REFERENCES**

* 1. Lehman, Meir M. (1980). "Programs, Life Cycles, and Laws of Software Evolution". Proc. IEEE 68 (9): 1060–1076.
  2. Eirini Kalliamvakou et al, The Promises and Perils of Mining GitHub, MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 92-101
  3. "GitHub Press Info". github.com. GitHub. Retrieved 2015-03-30.
  4. Georgios Gousios et al, An Exploratory Study of the Pull-based Software Development Model ,MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 384-387
  5. Lehman, M. M. (1980). "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle". Journal of Systems and Software 1: 213–221.
  6. Liguo Yu and Alok Mishra (2013) An Empirical Study of Lehman’s Law on Software Quality Evolution in International Journal of Software and Informatics, 11/2013
  7. Jyoti Sheoran et al, Understanding "watchers" on GitHub, MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 336-339
  8. Jesus M. Gonzalez-Barahona et al, Studying the laws of software evolution in a long-lived FLOSS Project, JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS J. Softw. Evol. and Proc. 2014; 26:589–612
  9. Xu Ben, Shen Beijun,Yang Weicheng, Mining Developer Contribution in Open Source Software Using Visualization Techniques, Intelligent System Design and Engineering Applications (ISDEA), 2013 Third International Conference, 16-18 Jan. 2013, 934 – 937
  10. Ioannis Skoulis et al, Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?, 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014), At Thessaloniki, Hellas
  11. M.M. Mahbubul Syeed, Imed Hammouda, Tarja Syst¨a, Evolution of Open Source Software Projects: A Systematic Literature Review, Journal of Software, Vol 8, No 11 (2013), 2815-2829, Nov 2013
  12. Hudson Borges, Marco Tulio Valente, Andre Hora, Jailton Coelho, On the Popularity of GitHub Applications: A Preliminary Note
  13. Tegawende F. Bissyand ´ e, Got Issues? Who Cares About It? An Investigation of Issue Trackers of 105 Projects, DOI: 10.1109/ISSRE.2013.6698918 Conference: Software Reliability Engineering
  14. Language Trends on GitHub, https://github.com/blog/2047-language-trends-on-github, August 19, 2015
  15. WG\_LCP - Working Group for Life Cycle Processes, Oversight Committee: C/S2ESC - Software & Systems Engineering Stand, 14764-2006 - ISO/IEC International Standard for Software Engineering - Software Life Cycle Processes – Maintenance
  16. <https://developer.github.com/v3/activity/starring/>
  17. Yan Dong, Shahab Mohsen, Does Firefox obey Lehman’s Laws of software Evolution?
  18. Guowu Xie et al, Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software, Software Maintenance, 2009. ICSM 2009. IEEE International Conference on 20-26 Sept. 2009
  19. Iulian Neamtiu et al, Towards a better understanding of software evolution: an empirical

study on open-source software, JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS

J. Softw.: Evol. and Proc. 2013; 25:193–218